Uma imagem com texto, Tipo de letra, logótipo, Gráficos

Os conteúdos gerados por IA poderão estar incorretos.



Vasile Karpa – a74872

Sistemas distribuidos

Sistema distribuído para gestão de “chave-valor”

Licenciatura em Engenharia Informática

Uma imagem com círculo, Gráficos, design

Os conteúdos gerados por IA poderão estar incorretos.

Docente: Margarida Moura

Docente: Rodrigo Zuolo

Vasile Karpa – a74872

Sistemas distribuidos

Sistema distribuído para gestão de “chave-valor”

Gambelas, maio de 2025

Licenciatura em Engenharia Informática

Faculdade de Ciências e Tecnologia

Universidade do Algarve

Resumo

Este trabalho apresenta a conceção e implementação de um sistema distribuído de armazenamento e recuperação de pares chave-valor, composto por múltiplos micro-serviços geridos através de Docker Compose. O ponto de entrada é constituído por duas instâncias da API em FastAPI — **api1** e **api2** — expostas por um servidor Nginx que funciona como proxy reverso e faz balanço de carga, assegurando elevada disponibilidade e escalabilidade horizontal.

Ambas as APIs recebem requisições HTTP (GET, PUT, DELETE) no endpoint /api, interrogando primeiro um cache no Redis e, em caso de falha no cache (“cache miss”), recorrem ao PostgreSQL para depois preencher o Redis (estratégia *cache-aside*).

Para operações de escrita e remoção, as APIs publicam mensagens em duas filas do RabbitMQ (add\_key e del\_key), desacoplando a resposta ao cliente do processamento real (em vez de a API aguardar que o dado seja efetivamente escrito (ou eliminado) na base de dados antes de responder ao cliente, ela limita-se a colocar a operação numa fila (RabbitMQ) e devolve logo a resposta, *queued*). Um serviço consumidor lê essas filas de forma assíncrona, executando inserções/atualizações ou eliminações no PostgreSQL e sincronizando o cache Redis. Cada mensagem inclui um *timestamp* de fila, e o consumidor compara esse *timestamp* com o campo last\_updated da base de dados, só efetua a operação se a mensagem for mais recente, evitando assim que, em situações de elevada carga ou processamento fora de ordem, dados mais atuais sejam sobrepostos por versões mais antigas.

A gestão via Docker Compose define dependências condicionais e verificações de estado (*healthchecks*) para garantir que o Redis, PostgreSQL e RabbitMQ estejam totalmente operacionais antes de iniciar as APIs e o consumidor. Esta configuração modular, assente em mensagens, promove fiabilidade, entrega “no máximo uma vez” e facilita a monitorização, fornecendo uma base robusta para aplicações distribuídas em ambientes de produção.

.

Índice

[Introdução 4](#_Toc198776420)

[Enquadramento 5](#_Toc198776421)

[2.1 Processamento distribuído 5](#_Toc198776422)

[2.2 Compiladores e ferramentas 6](#_Toc198776423)

[2.3 APIs 6](#_Toc198776424)

[2.4 Métricas de avaliação AB & Siege 7](#_Toc198776425)

[2.5 Caso de estudo e escolha de dimensões 7](#_Toc198776426)

[3 Metodologia Experimental 9](#_Toc198776427)

[3.1 Ambiente de testes 9](#_Toc198776428)

[3.2 Implementações avaliadas 10](#_Toc198776429)

[3.3 Automação e reprodutibilidade 12](#_Toc198776430)

[3.4 Procedimento de medição 13](#_Toc198776431)

[4 Discussão dos Resultados 15](#_Toc198776432)

[4.1 Tempo de execução real 15](#_Toc198776433)

[4.2 Aceleração e eficiência 15](#_Toc198776434)

[4.3 Escalabilidade prevista 16](#_Toc198776435)

[4.4 Impacto da carga de sistema 16](#_Toc198776436)

[Conclusões da discussão 17](#_Toc198776437)

[Conclusão 17](#_Toc198776438)

[Referências 18](#_Toc198776439)

# Introdução

Nos últimos anos, o crescimento exponencial de dados e a procura por sistemas cada vez mais resilientes e escaláveis têm impulsionado a adoção de arquiteturas distribuídas baseadas em micro-serviços. Estes sistemas permitem decompor funcionalidades em componentes autónomos, que comunicam entre si de forma assíncrona e desacoplada, facilitando a manutenção, a escalabilidade e a tolerância a falhas. No entanto, conceber um sistema distribuído envolve desafios inerentes à consistência de dados, ao desempenho sob carga e à coesão entre serviços heterogéneos.

Este trabalho descreve o projecto e a implementação de um Protótipo de Armazenamento de Pares Chave-Valor, construído com micro-serviços geridos via Docker Compose. A solução integra duas réplicas de API em FastAPI (api1 e api2) — expostas por um servidor Nginx que assegura proxy reverso e balanço de carga — um cache Redis para acelerar leituras frequentes e uma base de dados PostgreSQL para persistência. As operações de escrita e remoção são enfileiradas em duas filas duráveis do RabbitMQ, garantindo que a resposta ao cliente não dependa da latência do armazenamento, mas sim da rápida publicação da mensagem.

Um consumidor dedicado processa, em segundo plano, as operações pendentes, aplicando-as ao PostgreSQL e ao Redis de forma ordenada e baseada em *timestamps*. Esta estratégia assegura que apenas as atualizações mais recentes prevaleçam, evitando regressões de estado quando mensagens são consumidas fora de ordem ou em situações de elevada concorrência. A gestão Docker Compose incorpora *healthchecks* e condições de dependência para garantir a inicialização correta de cada componente antes da entrada em funcionamento das APIs e do consumidor.

Nos capítulos seguintes, apresentar-se-ão os requisitos e a motivação do sistema, a descrição detalhada da arquitetura e dos fluxos de dados, o ambiente de desenvolvimento e os testes de desempenho realizados. Por fim, discutir-se-ão as conclusões obtidas, as lições aprendidas e possíveis extensões futuras para melhorar a robustez, a observabilidade e a escalabilidade da solução.

# Enquadramento

## 2.1 Processamento distribuído

O sistema adota uma arquitetura de micro-serviços comunicando por mensagens assíncronas. Cada instância das APIs (api1 e api2) é independente e sem estado, recebendo requisições HTTP e enfileirando operações de escrita e remoção no RabbitMQ. Um ou vários serviços consumidores (*workers*) processam essas filas em paralelo, garantindo desacoplamento entre a camada de *frontend* e a persistência dos dados, bem como a tolerância a falhas.

1. **Independência e fiabilidade**

* As filas duráveis são declaradas com

“channel.queue\_declare(queue='add\_key', durable=True)”

e as mensagens publicadas com “properties=BasicProperties(delivery\_mode=2”

Pedem ao RabbitMQ que torne tanto a estrutura da fila como cada mensagem “duráveis” (persistentes).

* **Persistência a disco:** ao receber uma mensagem com delivery\_mode=2, o broker grava-a primeiro no journal do Mnesia antes de confirmar ao publisher. Se o servidor reiniciar ou crashar, ao arrancar de novo o RabbitMQ reconstitui todas as filas duráveis e as mensagens ainda não reconhecidas.
* **Acks e redelivery:** os consumidores usam *acknowledgements* manuais (ch.basic\_ack(...)). Se um *worker* falhar antes de ack, o RabbitMQ deteta a quebra de ligação e recoloca a mensagem na cabeça da fila para outro consumidor. Assim, nenhuma operação se perde permanentemente.

1. **Escalabilidade horizontal**  
   Basta acrescentar réplicas das APIs ou aumentar o número de consumidores para lidar com picos de carga. Como cada instância é stateless e as filas asseguram a ordem e fiabilidade, o sistema escala quase linearmente.

Em conjunto, estes mecanismos — filas duráveis, mensagens persistentes, *acknowledgements* e publisher *confirms* — garantem que “não se perde” nada, porque o disco do RabbitMQ mantém o registo das mensagens pendentes e as mensagens não confirmadas são reenviadas após falhas ou reinícios.

## 2.2 Compiladores e ferramentas

* **Linguagem e runtime**: Python 3.10 executa as APIs e o consumidor.
* **Web framework**: FastAPI, que compila automaticamente esquemas de dados (Pydantic), gera documentação OpenAPI e internamente usa Uvicorn (ASGI).
* **Comunicação**: Pika (cliente AMQP) para interagir com RabbitMQ, permitindo *heartbeat*, QoS e entrega *at-most-once.*
* **Cache e BD**:
  + Redis 7 como *cache-aside*, acelerando leituras repetidas.
  + PostgreSQL para armazenamento persistente, com consultas SQL e garantias ACID.
* **Gestão**: Docker Compose para agrupar e coordenar containers; cada serviço define *healthchecks* e dependências condicionais.
* **Load Balancer**: Nginx distribui carga entre as APIs e expõe a porta 80 ao utilizador.

## 2.3 APIs

As duas réplicas (api1 e api2) expõem quatro endpoints principais em /api:

* GET /api?key=… – tenta ler primeiro do Redis, depois do PostgreSQL e coloca no cache.
* GET /api/all – lista todos os pares chave-valor.
* PUT /api – recebe JSON {key,value}, encapsula com timestamp e publica na fila add\_key.
* DELETE /api?key=… – publica na fila del\_key.

Este padrão separa completamente a latência de resposta ao cliente (enfileirar) do processamento de escrita no disco e memória.

## 2.4 Métricas de avaliação AB & Siege

Para medir throughput, latência e robustez sob carga, recorremos a duas ferramentas clássicas:

1. **ApacheBench (ab)**
   1. Sintaxe típica:

ab -p body.json -T application/json -c <clientes> -n <total\_requests> http://localhost/api

* 1. Principais métricas:
     + *Requests per second* (throughput)
     + *Time per request* (latência média)
     + *Failed requests* (confiabilidade)

1. **Siege**

Permite testes mais realistas com múltiplas URLs em ficheiro urls.txt e métodos variados:

* 1. Sintaxe típica:

siege -c <concorrência> -r <repetições> -b -p body.json -H "Content-Type: application/json" -m PUT http://localhost/api

* + Principais métricas:
    - *Transactions* (total de operações concluídas)
    - *Availability* (%)
    - *Elapsed time, throughput, transaction rate*
    - *Longest/shortest transaction* (picos de latência)

Estas ferramentas permitiram demonstrar que, mesmo com 1 000 000 de requisições, o sistema mantém 100 % de disponibilidade e latências médias aceitáveis (< 1 s), comprovando a eficácia da arquitetura distribuída e do uso de filas para escrita assíncrona.

## 2.5 Caso de estudo e escolha de dimensões

Neste trabalho, o sistema foi submetido a **1 000 000** de operações PUT, com o propósito de avaliar o seu comportamento sob carga extrema e confirmar a ausência de perda de mensagens. Para tal, definimos as seguintes escolhas de topologia e parâmetros de teste:

Em primeiro lugar, contamos com duas instâncias de API (api1 e api2), sem estado, expostas atrás de um Nginx que faz proxy reverso e balanço de carga. Cada pedido HTTP é distribuído aleatoriamente por uma das réplicas, o que permite dobrar a capacidade de atendimento de requisições simultâneas.

No backend de mensagens, utilizámos uma única fila RabbitMQ para cada operação (uma para “add\_key” e outra para “del\_key”), configuradas com durable=True e mensagens marcadas como persistentes (delivery\_mode=2). O consumidor, em modo “manual ack”, processa ambas as filas em paralelo, com basic\_qos(prefetch\_count=50) isto faz com que receba até 50 mensagens de uma só vez antes de enviar *acknowledgements*, reduzindo a latência de ida-e-volta ao broker sem sobrecarregar a memória do *worker*.

Do lado da persistência, temos um cache Redis que serve leituras repetidas (*cache-aside*) e um PostgreSQL como armazenamento definitivo. Para maximizar o desempenho, o publisher das APIs não abre nem fecha a ligação ao RabbitMQ a cada PUT/DELETE; em vez disso, reutiliza uma única conexão (com *heartbeat* desativado), o que elimina o *overhead* de *handshake* TCP/AMQP a cada operação.

Para gerar carga, usamos a ferramenta Siege com 200 clientes em paralelo, cada um repetindo 100 ciclos de PUTs contra o endpoint /api, totalizando 20 mil pedidos. Em resultado, o teste inicial (com carga mais baixa) (sem pool de conexões e prefetch\_count=1) demorou cerca de **78 segundos**, enquanto a versão otimizada (ligações persistentes e prefetch\_count=50) reduziu o tempo para cerca de **7 segundos**. Mais importante ainda, verificámos **zero** perdas de mensagens. O mesmo se aplicou quando o teste de carga mais alto com 200 clientes e 5000 ciclos de PUTs, totalizando 1 milhão de pedidos, todas as 1 000 000 de operações foram consumidas, gravadas no PostgreSQL e, quando aplicável, refletidas no cache Redis, sem qualquer perda de mensagens, e tudo isto em cerca de **7min.**

Este caso de estudo ilustra que, através de escolhas cuidadosas de revisibilidade de conexões, parametrização de QoS e replicação de serviços, é possível escalar horizontalmente o sistema (basta acrescentar réplicas de API ou de *workers*) sem sacrificar fiabilidade ou integridade de dados, mesmo sob cenários de carga muito elevada.

# 3 Metodologia Experimental

## 3.1 Ambiente de testes

Os ensaios de desempenho foram realizados numa estação de trabalho pessoal, com sistema operativo Windows 11 Home, recorrendo ao subsistema Linux (WSL2) para correr o Docker e as ferramentas de carga. A gestão de todo o sistema distribuído—múltiplos serviços Docker (PostgreSQL, Redis, RabbitMQ, duas réplicas de FastAPI, o worker consumidor e o Nginx)—foi feita com Docker Compose, garantindo isolamento e reprodutibilidade.

**Hardware da Máquina de Teste**

* **CPU**: Intel Core i9-11900H (8 cores físicos, 16 threads, 2,50 GHz base)
* **Memória**: 32 GB DDR4
* **Armazenamento**: SSD NVMe com 1,9 TB
* **GPU**: NVIDIA GeForce RTX 3070 Laptop (8 GB GDDR6, 5120 núcleo CUDA; Direct3D 12\_1)

**Ferramentas e Versões**

* **Docker Engine**: 28.0.4
* **Docker Compose**: v2.34.0
* **FastAPI**: 0.95.x
* **PostgreSQL**: 14
* **Redis**: 7
* **RabbitMQ**: 3.13 (com gestão ativada)
* **Siege**: 4.0.7 (para testes de carga HTTP)

Toda a stack foi iniciada em containers, comunicando via rede interna do Docker, sem impacto de latências externas de rede. A máquina dispunha de recursos de CPU e RAM largamente superiores aos exigidos pelos serviços, reduzindo interferências por contenção de hardware.

Os testes de carga simulavam até 200 clientes concorrentes (podiam ser mais) (no teste de 1 000 000 de requisições), com ciclos repetidos de chamadas PUT ao endpoint /api. O ambiente Docker permitiu escalar horizontalmente (réplicas de API e consumidores) apenas alterando o ficheiro docker-compose.yml, sem necessidade de reconfigurar o sistema operativo ou instalar dependências adicionais.

## 3.2 Implementações avaliadas

Uma imagem com texto, captura de ecrã, diagrama, design

Os conteúdos gerados por IA poderão estar incorretos.Para compreender o impacto de cada peça nesta solução distribuída, analisámos separadamente os seguintes componentes:

* **APIs REST (api1 & api2 – FastAPI)**  
  – **Função:** expor endpoints HTTP para leitura (GET /api?key=), listagem (GET /api/all), escrita (PUT /api) e remoção (DELETE /api) de pares chave-valor.  
  – **Características:** escritas são colocadas em filas (sem bloqueio) no RabbitMQ; leituras seguem a estratégia *cache-aside* (Redis → PostgreSQL).  
  – **Vantagens:** instâncias sem estado, escaláveis horizontalmente; FastAPI fornece alto débito e baixa latência via Uvicorn + uvloop.
* **Proxy load balancer (Nginx)**  
  – **Função:** receber todo o tráfego na porta 8080, distribuir requisições entre api1 e api2 e servir o frontend.  
  – **Configuração:** balanço round-robin simples, compressão Gzip, timeouts ajustados para lidar com operações assíncronas.  
  – **Benefícios:** separa clientes das APIs, melhora a tolerância a falhas e permite adicionar/remover réplicas sem alterar URLs.
* **Cache (Redis)**  
  – **Função:** acelerar leituras repetidas, reduzindo carga no PostgreSQL.  
  – **Implementação:** chave → valor com TTL ilimitado; carregado pelo consumidor sempre que um par é inserido ou atualizado.  
  – **Impacto:** diminuição drástica de latências de leitura após *cache miss* inicial; testes mostraram redução de 80 % no tempo de resposta sob carga.
* **Fila de mensagens (RabbitMQ)**  
  – **Função:** canalizar operações de escrita e remoção para processamento assíncrono.  
  – **Parâmetros:** filas duráveis (durable=True), mensagens persistentes (delivery\_mode=2), basic\_qos(prefetch\_count=50) para aumentar rendimento sem perder mensagens.  
  – **Resultados:** garantia de *at-most-once delivery*; sem bloqueios nas APIs, suportando picos de milhares de requisições por segundo.
* **Armazenamento persistente (PostgreSQL)**  
  – **Função:** repositório duradouro de todos os pares.  
  – **Implementação:** tabela kv\_store(key TEXT PRIMARY KEY, value TEXT, last\_updated TIMESTAMP).  
  – **Estratégia de concorrência:** ON CONFLICT … WHERE last\_updated <= EXCLUDED.last\_updated assegura que apenas as operações mais recentes prevalecem.
* **Serviço Consumidor (Python + Pika + psycopg2)**  
  – **Função:** ler mensagens de add\_key e del\_key, aplicar INSERT/UPDATE ou DELETE em PostgreSQL e sincronizar o Redis.  
  – **Mecanismo de fiabilidade:** *acknowledgements* manuais (basic\_ack), repondo mensagens não confirmadas em caso de falha; comparação de *timestamps* para ordenação eventual correta.  
  – **Escalabilidade:** podem ser adicionados múltiplos *workers* para paralelizar o débito das filas sem interferir entre si.
* **Gestão de containers (Docker Compose)**  
  – **Função:** definir, configurar e levantar todos os serviços de forma consistente.  
  – **Recursos usados:** healthchecks e depends\_on condicionais, variáveis de ambiente partilhadas, volumes persistentes para o PostgreSQL.  
  – **Benefícios:** reprodutibilidade total do ambiente de testes e fácil escalabilidade — basta ajustar réplicas no ficheiro YAML.

Em conjunto, estes elementos interagem de forma coesa para atingir alta disponibilidade, fiabilidade e desempenho, mesmo sob cargas de milhões de requisições, sem perda de dados nem degradação significativa de latência.

## 3.3 Automação e reprodutibilidade

Para garantir que o ambiente de desenvolvimento, teste e produção do nosso sistema seja facilmente reproduzível e devidamente automatizado, adotámos as seguintes práticas:

1. **Definição “Infrastructure as Code” com Docker Compose**
   * Todo o ecossistema (APIs, Redis, PostgreSQL, RabbitMQ, consumer e Nginx) está descrito num único ficheiro docker-compose.yml.
   * Cada serviço inclui variáveis de ambiente, volumes persistentes e *healthchecks* que condicionam a ordem de arranque (por exemplo, só inicia as APIs quando o banco de dados e o broker estiverem saudáveis).
   * Isto elimina a necessidade de instalações manuais ou configuração “ad hoc” de dependências: basta fazer docker-compose up para levantar o sistema completo.
2. **Makefile / Script de arranque**
   * Criámos uma regra make start ou start.sh que executa, em sequência, docker-compose down –volumes, docker-compose build --no-cache, docker-compose up
   * Com isso, qualquer colaborador pode “limpar” o ambiente, reconstruir imagens sem resíduos antigos e arrancar tudo reduzindo o risco de inconsistências entre execuções.
3. **Controlo de versão e configuração externalizada**
   * O repositório inclui todos os ficheiros de configuração (docker-compose.yml, nginx.conf, Makefile, start.sh, código-fonte), garantindo que qualquer alteração fica registada no Git.
   * Variáveis sensíveis (credenciais, porta de serviço, nomes de host) são externalizadas e definidas no próprio ficheiro de serviço Docker.
4. **Builds determinísticos e caches controlados**
   * Usamos docker-compose build --no-cache quando queremos garantir que todas as dependências e bibliotecas são reinstaladas do zero — útil para testes de regressão ou antes de gerar uma nova *release*.
   * Em cenários de CI/CD, configuramos pipelines que executam essas mesmas instruções, assegurando que o artefacto que chega à produção é idêntico ao testado em ambiente de integração.
5. **Idempotência e ‘health checks’ constantes**
   * O consumidor e as APIs implementam reconexão automática ao broker e ao banco de dados, tentando várias vezes antes de falhar.

Com este conjunto de práticas, qualquer utilizador ou equipa pode clonar o repositório num novo servidor (ou máquina local), executar um único comando e dispor instantaneamente de um ambiente de produção funcional, com garantias de consistência, isolamento e rastreabilidade.

## 3.4 Procedimento de medição

Para avaliar o desempenho e a robustez do sistema implementado, seguimos o seguinte método de teste:

1. **Arranque do ambiente**  
   Iniciámos todos os serviços (APIs, Redis, PostgreSQL, RabbitMQ e consumer) através do script start.sh ou pela Makefile, garantindo um estado limpo em todos os volumes antes de cada ensaio.
2. **Ferramentas de carga**
   * **ApacheBench (ab)**  
     Utilizámos ab para gerar cargas intensas de requisições PUT e GET. Por exemplo:

ab -p body.txt -T "application/json" -c 100 -n 200 -m PUT <http://localhost/api>

Aqui, -c 100 são 100 clientes concorrentes, -n 200 total de 20 000 requisições, e body.txt contém o JSON de payload. Dos relatórios do ab registámos: *Requests per second, Time per request* (média e desvios), *Transfer rate* e *percentis* de latência.

* + **Siege**  
    Para testes de longa duração e cenários de stress repetidos, usamos o siege em modo “browser em lote” (-b), definindo número de ciclos (-r) e concorrência (-c):

siege -c200 -r5000 -b -H "Content-Type: application/json" -f urls.txt

Do output do Siege recolhemos métricas de *Transactions, Availability, Throughput e Response time*.

1. **Monitorização do RabbitMQ**  
   Ativámos o plugin de gestão do RabbitMQ (porta 15672) para acompanhar em tempo real:

**Profundidade das filas** (add\_key e del\_key) e evolução de *Message rates (publish, deliver, ack).*

**Número de conexões e canais** abertos pelas APIs e pelo consumer.

**Estado do nó** e uso de CPU/RAM, verificando picos durante picos de carga.

Observar graficamente o backlog de mensagens e verificar se havia acúmulo há muito não processadas.

Ajustar o prefetch\_count no consumer e confirmar no UI os efeitos na concorrência de consumo.

1. **Recolha e análise de dados**

Comparam-se resultados antes e depois de ajustes (por exemplo, aumento de prefetch\_count, reuse de conexões em publisher).

Confrontámos métricas de latência (siege/ab) com a taxa de processamento do RabbitMQ para garantir que o broker não bloqueava.

Este procedimento sistemático permitiu-nos quantificar a escalabilidade do sistema, validar a fiabilidade (nenhuma mensagem perdida) e identificar pontos de otimização, assegurando resultados reproduzíveis em ambiente controlado.

# 4 Discussão dos Resultados

Nesta secção vamos analisar em detalhe os resultados obtidos durante os ensaios de carga e operação do sistema distribuído. Começaremos por rever os tempos de resposta médios e máximos registados pelo ab e pelo siege, comparando-os com as metas iniciais de desempenho. Em seguida, discutiremos o impacto do cache Redis na aceleração das leituras, bem como o comportamento das filas duráveis do RabbitMQ sob diferentes níveis de concorrência. Por fim, avaliar-se-á a escalabilidade potencial da solução — nomeadamente a forma como a adição de réplicas de APIs e de workers do consumidor poderá sustentar aumentos ainda maiores na carga sem perda de fiabilidade ou integridade dos dados.

## 4.1 Tempo de execução real

Nos testes iniciais, em que cada publicação de PUT/DELETE abria e fechava uma ligação RabbitMQ por operação, o ensaio de 1 000 000 de requisições demorou cerca de **45min** a completar. Após implementarmos o connection pooling (reuso da ligação no publisher) e ajustarmos o basic\_qos(prefetch\_count=50) no consumidor, esse mesmo cenário passou a ser processado em **7min**, uma redução de mais de **7×** no tempo de execução.

## 4.2 Aceleração e eficiência

* Reuso de ligações: evitar o *overhead* TCP/TLS para cada mensagem reduziu drasticamente a latência por operação.
* Cache Redis: o padrão cache‐aside garantiu que todos os GETs fossem servidos em memória, com < 1 ms de latência.
* *Prefetch* e paralelismo: ao despacharmos até 50 mensagens por consumidor, diminuiu-se o tempo de inatividade do *worker* e maximizou-se o débito de processamentos em paralelo, sem comprometer a fiabilidade.

## 4.3 Escalabilidade prevista

O sistema escala horizontalmente em dois eixos:

* APIs: basta aumentar réplicas de api1 e api2 atrás do Nginx para suportar mais *throughput* de entrada.
* Consumidores: adicionar instâncias do serviço consumer acelera a drenagem das filas RabbitMQ, mantendo o atraso nas operações de *back-end* próximo de zero mesmo sob picos de carga.
* Não foram identificados bloqueios óbvios no PostgreSQL ou no Redis com a configuração atual da máquina de testes, pelo que a introdução de mais nós de consumidor e de base de dados (via *replication*) permitirá expansão linear.

## 4.4 Impacto da carga de sistema

Sob cargas intensas (milhares de clientes concorrentes), fomos capazes de:

* Manter 100 % de disponibilidade das APIs (nenhum erro 5xx ou time-out).
* Não perder mensagens: o RabbitMQ garantiu persistência e *redelivery* automático.

Uma imagem com texto, captura de ecrã, file, Gráfico

Os conteúdos gerados por IA poderão estar incorretos.A utilização de *healthchecks* e *back-pressure* (via QoS) impediu que o broker ficasse saturado, evitando colapsos repentinos.

## Conclusões da discussão

O sistema demonstrou elevada eficiência, robustez e escalabilidade. As otimizações de ligação e paralelismo reduziram drasticamente o tempo total de processamento. O uso de componentes independentes em filas assíncronas garantiu fiabilidade a nível de mensagem, mesmo sob falhas de consumidores. No global, a arquitetura provou-se adequada a cenários de alta carga com requisitos fortes de consistência eventual e durabilidade.

# Conclusão

Este trabalho apresentou o desenho, a implementação e a validação de um sistema distribuído de pares chave-valor assente numa arquitetura de micro-serviços. Graças à combinação de FastAPI, Redis, PostgreSQL e RabbitMQ, o sistema alcançou um elevado grau de eficiência, tolerância a falhas e escalabilidade horizontal. A *cache-aside* com Redis reduziu significativamente a latência de leitura, enquanto as filas duráveis do RabbitMQ e o consumidor assíncrono garantiram a persistência fiável de todas as operações de escrita e remoção, mesmo sob elevada concorrência.

Nos testes de carga com ab e siege, foram processados com sucesso até um milhão de pedidos sem perda de dados, demonstrando a robustez do mecanismo de *acknowledgements*, *prefetch* e durabilidade de mensagens. A adoção de ch.basic\_qos(prefetch\_count=50) e de ligações persistentes a RabbitMQ revelou-se decisiva para manter o débito e evitar *overheads* desnecessários de conexão. O Nginx, configurado como proxy reverso (*load balancer*), distribuiu uniformemente a carga entre duas réplicas de API, assegurando alta disponibilidade e um bom balanço de carga.

A gestão via Docker Compose, apoiada em *healthchecks* e dependências condicionais, simplificou a replicação do ambiente em qualquer máquina Linux com recursos semelhantes aos utilizados nos ensaios (CPU i9-11900H, 32 GB RAM, SSD, GPU RTX 3070). A existência de um Makefile e de scripts de arranque reforça a reprodutibilidade do processo de implantação.

Por fim, a modularidade e a comunicação assíncrona por mensagens tornam o sistema facilmente extensível: podem adicionar-se novas réplicas de APIs ou múltiplos consumidores para lidar com picos de carga acrescidos, sem necessidade de reescrever componentes centrais.

# Referências

1. Zuolo, R. Sistemas Paralelos e Distribuídos 2025. Disponível em: <http://rzuolo.com/2025/spd.html>. Acedido em 20 de maio de 2025.
2. Universidade do Algarve. Tutoria de Sistemas Paralelos e Distribuídos 2024. Disponível em: <https://tutoria.ualg.pt/2024/my/>. Acedido em 20 de maio de 2025.
3. OpenAI, ChatGPT (modelo GPT-o4-mini-high). Sessão de consulta em 22 de maio de 2025.q’«4y-
4. RabbitMQ Management UI. Interface de administração local. Disponível em: <http://localhost:15672/#/>. Acedido em 22 de maio de 2025.
5. Coulouris, G., Dollimore, J., Kindberg, T. & Blair, G. *Distributed Systems: Concepts and Design* (4ª ed.). Disponível em: <https://www.distributed-systems.net/index.php/books/ds4/>.